CS238 - Project 1 README

Shreya S Ramanujam

October 15, 2025

1 Description of Algorithm Used

For this project, I used the K2 algorithm followed by a greedy local search (add/remove edges greedily) to further optimize the graph we get from K2. The pseudocode for the algorithm is detailed below:

Algorithm 1: Bayesian Network Structure Learning using K2 and Greedy Hill Climbing

```
Input: Data matrix D with n discrete variables
Output: Directed acyclic graph G

Initialization:
    Initialize an empty directed graph G with n nodes.
    Set a variable order [1, 2, ..., n].
    Compute initial score of empty graph S ← BayesianScore(G, D).

K2 Edge Addition Phase:
    for i ← 1 to n do
        for j ← i + 1 to n do
            Add edge (X_i → X_j) to G.
            Compute S_new ← BayesianScore(G, D).
            if S_new > S then
                S ← S_new                                // Keep the edge
            else
                Remove edge (X_i → X_j)                  // Revert if score decreases

Greedy Hill Climbing Refinement:
    for each node pair (i, j) with i ≠ j do
        if there is no edge (i → j) in G then
            Add edge (i → j) to G.
            if G is acyclic then
                Compute S_new ← BayesianScore(G, D).
                if S_new > S then
                    S ← S_new
                else
                    Remove edge (i → j)
            else
                Remove edge (i → j)
        else
            Remove edge (i → j).
            Compute S_new ← BayesianScore(G, D).
            if S_new > S then
                S ← S_new
            else
                Re-add edge (i → j)

Output:
Return the final DAG G with the highest Bayesian score S.
```

For the K2 algorithm, I did try different modifications, like:

- Taking 10 random node orders, doing K2 on all of them and then picking the graph with the maximum Bayesian Score.
- Picking an edge which reduces the score a small fraction of the time (around 10% of the time).
- Trying a Mutual-Information informed ordering for the nodes, which means that nodes with a higher total Mutual Information will be earlier in the ordering.

Ultimately, I just settled on a regular node ordering for K2 (1, 2, ... n) and did local search afterwards on K2 graph. This seemed to give the best final Bayesian score.

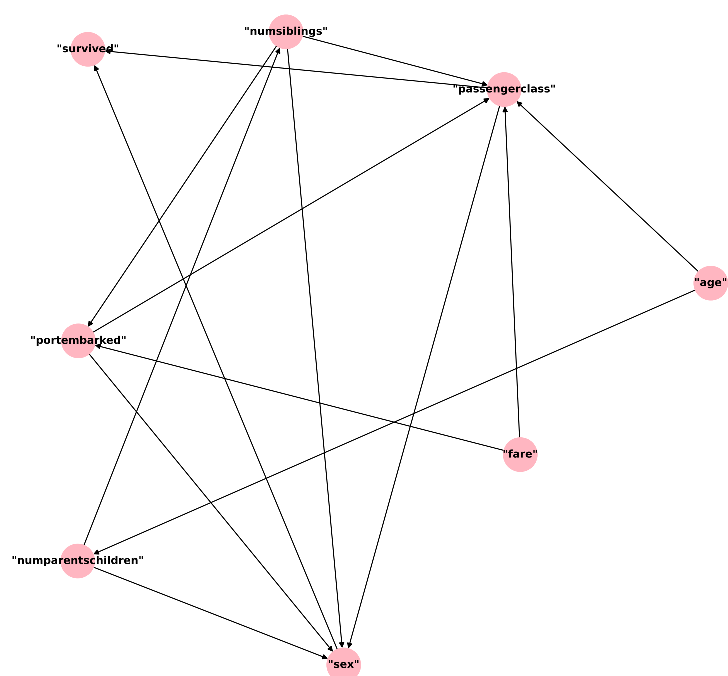
2 Running Times

The running time for various datasets is given below:

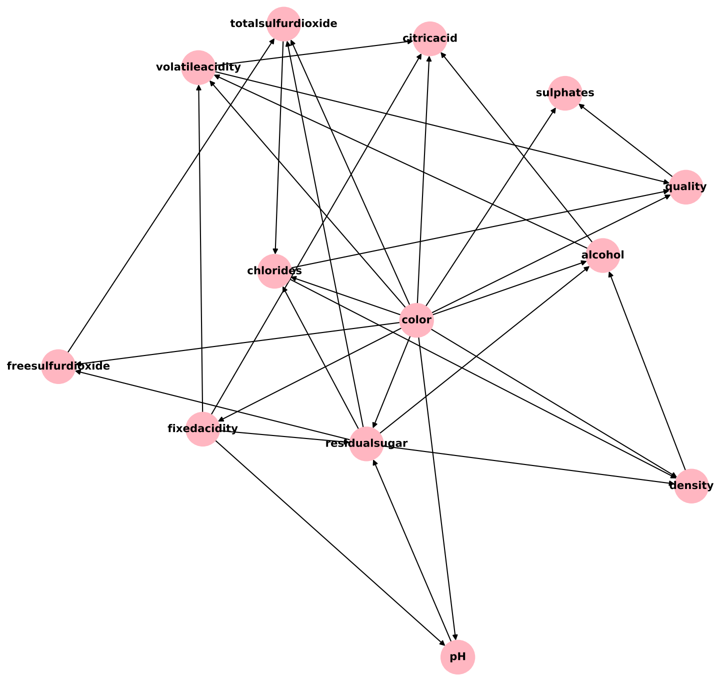
- **small.csv:** 0.25 seconds
- **medium.csv:** 11.07 seconds
- **large.csv:** 1145.86 seconds

3 Visualizing Graphs

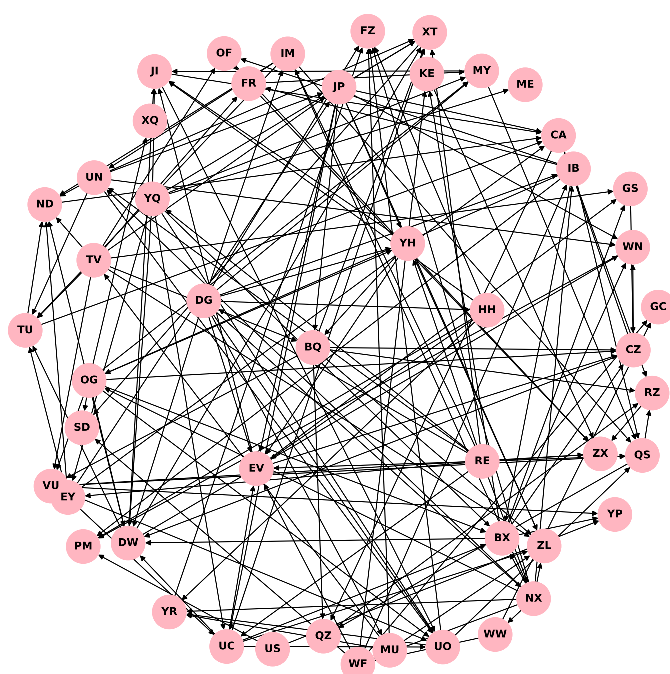
3.1 Final Graph for `small.csv`



3.2 Final Graph for `medium.csv`



3.3 Final Graph for `large.csv`



4 Code

First, we define all our helper functions in `utils.py`:

```python
# utils.py
import numpy as np
import math
from collections import defaultdict
import networkx
from sklearn.metrics import mutual_info_score

def count_configurations(node, parents, data):
    """
    Count occurrences of each configuration of a node given its parents.
    """
    # initialize a dictionary of dictionaries initialized to 0
    counts = defaultdict(lambda: defaultdict(int))
    # print("Counting configurations for node:", node, "with parents:", parents)
    for row in data:
        parent_values = tuple(row[parents]) if parents else ()
        node_value = row[node]
        # print("Row:", row, "Parent values:", parent_values, "Node value:", node_value)
        counts[parent_values][node_value] += 1
    return counts

def bayesian_score(graph, data):
    """
    Calculate the Bayesian score of a given graph structure based on the provided data.
    """

    score = 0.0
    for node, parents in graph:
        # print("Node:", node, "Parents:", parents)
        counts = count_configurations(node, parents, data)
        # count the number of unique values the node can take
        unique_node_values = set(data[:, node])
        for parent_values, val_dict in counts.items(): # all key value pairs - here it is key : (parent values), value : {node val: count}
            # print("Parent values:", parent_values, "Value counts:", val_dict)
            m_ij0 = sum(val_dict.values()) # total count of all values for this parent configuration
            for m_ijk in val_dict.values():
                score += math.lgamma(m_ijk + 1) - math.lgamma(1)  # Assuming uniform prior : alpha_ijk = 1 for all k
            score += math.lgamma(len(unique_node_values)) - math.lgamma(m_ij0 + len(unique_node_values))

    return score

def mutual_info_order(data):
    """
    Compute the mutual information between all pairs of variables and return an ordering based on it.
    We do this to get a better initial ordering for the K2 algorithm.
```

```
    """
    num_vars = data.shape[1]
    mi_matrix = np.zeros((num_vars, num_vars))

    # Calculate mutual information for each pair of variables in given dataset
    for i in range(num_vars):
        for j in range(i + 1, num_vars): # mutual information is symmetric
            mi = mutual_info_score(data[:, i], data[:, j])
            mi_matrix[i, j] = mi
            mi_matrix[j, i] = mi

    # Sum mutual information for each variable
    mi_sums = np.sum(mi_matrix, axis=1)
    # Get ordering based on ascending mutual information sums
    order = np.argsort(mi_sums).tolist()

    return order
```

The actual data processing and structure learning algorithm are defined in `project1.py`. We call helper functions from `utils.py` wherever required.

```
# project1.py
import sys
import math
import numpy as np
from utils import bayesian_score, mutual_info_order
import networkx
import time


def write_gph(dag, idx2names, filename):
    with open(filename, 'w') as f:
        for edge in dag.edges():
            f.write("{}, {}\n".format(idx2names[edge[0]], idx2names[edge[1]]))


def compute(infile, outfile):
    # converting data csv to numpy array
    data = np.loadtxt(infile, delimiter=',', skiprows=1, dtype=int)

    # mapping variable names to indices to make computation faster (no more dicts)
    num_vars = data.shape[1]
    var_names = np.loadtxt(infile, delimiter=',', dtype=str, max_rows=1)
    name2idx = {var_names[i]: i for i in range(num_vars)}
    idx2names = {i: var_names[i] for i in range(num_vars)}

    # deciding an order for K2 algorithm
    order = list(range(num_vars))
    # initialize graph with no edges
    dag = networkx.DiGraph()
    dag.add_nodes_from(range(num_vars))
    # get the graph for the initial (no edges) structure
    graph = [(i, list(dag.predecessors(i))) for i in range(num_vars)]
    current_score = bayesian_score(graph, data)

    # time the algorithm
```

```python
    start_time = time.time()

    # K2 algorithm
    for i in range(num_vars):
        node = order[i] # for each node in the order, try to add right children
        for potential_child in order[i+1:]: # only consider nodes that come after it in the order
            dag.add_edge(node, potential_child) # add the edge
            new_graph = [(j, list(dag.predecessors(j))) for j in range(num_vars)]
            new_score = bayesian_score(new_graph, data)
            if new_score > current_score: # if score improves, keep the edge and update score
                current_score = new_score
            else: # otherwise remove the edge
                dag.remove_edge(node, potential_child)

    # now, take this graph and do greedy hill climbing to improve it further
    for i in order:
        for j in range(num_vars):
            if i == j: # self loops not allowed
                continue
            # try adding edge i -> j if it doesn't create a cycle
            if not dag.has_edge(i, j):
                dag.add_edge(i, j)
                if networkx.is_directed_acyclic_graph(dag): # only keep if no cycle
                    new_graph = [(k, list(dag.predecessors(k))) for k in range(num_vars)]
                    new_score = bayesian_score(new_graph, data)
                    if new_score > current_score:
                        current_score = new_score
                    else:
                        dag.remove_edge(i, j)
                else:
                    dag.remove_edge(i, j)
            else:
                # try removing edge i -> j
                dag.remove_edge(i, j)
                new_graph = [(k, list(dag.predecessors(k))) for k in range(num_vars)]
                new_score = bayesian_score(new_graph, data)
                if new_score > current_score:
                    current_score = new_score
                else:
                    dag.add_edge(i, j)

    end_time = time.time()
    print("Time taken: {:.2f} seconds".format(end_time - start_time))
    print("Final score: {}".format(current_score))
    write_gph(dag, idx2names, outfile)
    print("Wrote graph to {}".format(outfile))


def main():
    if len(sys.argv) != 3:
        raise Exception("usage: python project1.py <infile>.csv <outfile>.gph")

```

```
    inputfilename = sys.argv[1]
    outputfilename = sys.argv[2]
    # time the compute function
    compute(inputfilename, outputfilename)

if __name__ == '__main__':
    main()
```